

COMPASS

COMPILER ASSEMBLER PACKAGE

INTEGER BASIC COMPILER FOR THE TIMEX SINCLAIR 2068

by John Coffey

Section I

BASIC, MACHINE CODE, AND COMPILERS

BASIC is language already built into the TIMEX SINCLAIR 2068 computer. It stands for Beginner's All Purpose Symbolic Instruction Code. It allows you to communicate with the computer with words that resemble English and it is absolutely essential that you know how to use BASIC if you are going to use the compiler. The 2068 user manual is an excellent tutorial of BASIC then you may want to consult it.

When you write programs in Basic, however, you are not directly telling the computer what to do. The "brain" of the computer is the microprocessor and it is only able to understand numbers. For example, consider the following two items:

BASIC
GOTO 100

MACHINE CODE
195 20 234

The former is more easily understood by people and the latter is more easily understood by the microprocessor. But they are actually the same instruction in two different languages.

What brings people and microprocessors together is a form that they can communicate is the INTERPRETER built into the 2068 computer.

Unfortunately the interpreter slows everything down. Although BASIC programs run pretty quickly, much faster results can be obtained if the computer is programmed directly in its own language. An ASSEMBLER is a program that will let you program directly and advanced programmers use them all the time.

But assemblers are a great deal harder to use than BASIC. So what is needed is an alternative that is both simple and fast.

A COMPILER is exactly that kind of alternative. It changes some BASIC commands into very fast machine code. Compiled code is not as fast as assembled code, but it is a great deal easier to use.

You may now want to proceed to sections II and VII. If you try some of the sample programs, then you may get a good feel for how to use the compiler.

Section II

HOW TO USE THE COMPILER

- STEP 1: Write a program in BASIC using the rules given in section III and IV of these instructions. Then RUN it and make sure it works. If corrections have to be made, then RUN the program again when you have made them.
- Note that a program may not always RUN the same in BASIC as it will in machine code.
- STEP 2: Decide where you want to put the machine code. If the machine code is going to be less than 4000 bytes then my favorite place to put it is location 60000. This is because it can be easily typed 6E4. (Scientific notation).
- STEP 3: Use CLEAR to reserve a place for the machine code. For example, if you were to use location 6E4, then you would type CLEAR 59999, which is the next lower location.
- STEP 4: Type "RUN 9000". It is not a good idea to use GOTO because already dimensioned strings can mess up the compiler.
- STEP 5: The computer will ask where you want to put the machine code. If you are using my example, then you would use 6E4.
- STEP 6: The compiling will take a few minutes at most. The computer has to make two passes through the program to get all the GOTO and GOSUB'S right. The location of machine code for each line will be displayed on the left. This could be useful if you have to jump to a specific line of machine code from BASIC or other machine code.
- STEP 7: The compiler will tell you how many bytes are used by the compiled code. This, however, does not include the bytes taken by the dimensioned strings which are stored at the end of the compiled code.
- STEP 8: If you did not adequately follow the rules in section III and IV then the compiler will either stop with an error report or produce faulty code. Faulty code may crash the computer when run so it is a good idea to have a tape back-up of your BASIC program. Faulty code will usually not crash, but produces bad results instead.
- STEP 9: Call your machine code with: PRINT USR address. Since I almost always use location 6E4 then I almost always type: PRINT USR 6E4. It is not a good idea to use RAND USR address because printing will be disabled. If you use PAUSE USR address then everything will be printed to the lower half of the screen.

Section III

Compiler Syntax Table

The following terms are referred to in the table below. It is important to use in the right places the right type of argument, whether it be a number, variable or expression.

A number is any integer constant from 0 to 65535.

A variable is any single letter A to Z. It makes no difference if the letter is upper or lower case.

An item is either a number or a variable. An item can also be a number or variable preceded by PEEK, USR, or IN. An item may also be CODE with a one character string following it.

Expressions, abbreviated "expr", can consist of an unlimited series of items separated each pair by the following operators: +, -, *, /, <, =, AND, OR. No other operators are provided.

item AND item		LPRINT	(used same as PRINT)
AT expr, expr	(after PRINT)	item OR item	
BEEP expr, expr		OUT expr, expr	
BORDER expr		OVER expr	
BRIGHT expr		OVER item	(after PRINT)
BRIGHT item	(after PRINT)	PAPER expr	
CHR\$ item		PAPER item	(after PRINT)
CLS		PAUSE expr	
CODE "A"	(one letter only)	PEEK item	
CODE A\$ (item)	(any string)	PLOT expr, expr	
:		POINT (item, item)	
:	(after PRINT)	POKE expr, expr	
:	(after PRINT)	PRINT	
COPY		PRINT "PUT MESSAGE HERE"	
DIM A\$ (number)	(any string)	PRINT STR\$ number (STR\$ not optional)	
GOTO number		(other things may follow PRINT)	
GOSUB number		RAND expr (will also pass a value to	
IF expr THEN		the BC register pair.)	
IN item		REM	
INK expr		RETURN	
INK item		RND* 65536	
INKEY\$	(after PRINT)	:	(after PRINT)
INVERSE expr		SOUND expr, expr; etc.	
INVERSE item	(after PRINT)	STOP	
LET A= expr	(any variable)	STR\$ item	(after PRINT only)
LET A\$= "MESSAGE"	(any string)	TAB item, item	(after PRINT)
LET A\$ (item)=B\$(item)	(any string)	THEN	(after IF)
LET A\$ (item)=CHR\$ item		USR item (Passes no values to the BC	
LET A\$ (item)=INKEY\$		register, but will return	
item * item		the BC register pair.)	
item / item			
item + item			
item - item			
item = item			
item < item			

Section IV

LIMITATIONS

People have asked me if the compiler would compile itself. (Can the compiler itself be converted to machine code by its own actions?) This is definately not the case since the compiler uses a subset of BASIC. So there are limitations on how you can write your BASIC program if you want it converted to machine code. Experienced machine code programmers probably won't find this a problem, but novice programmers will have to get used to the lack of conveniences normally found in BASIC.

1. Line numbers should be from 1 to 8998.
2. Put only numbers after GOTO or GOSUB.
3. STOP will not work from inside a subroutine.
4. The execution of machine code must finish with a STOP statement.
5. The machine code only uses integers for math. This is especially noticeable when division is used. The following will work differently from machine code than BASIC:

```
10 LET N=10/3
20 PRINT STR$ N
```

BASIC

MACHINE CODE

3.33333

3

6. As in the above example, if you want to print a number, you must put STR\$ in front of it.
7. You can use anything after a PRINT statement that you normally would, except numbers or math expressions.

Illegal: 10 PRINT 2*2

Legal: 10 LET A=2*2

20 PRINT STR\$ A

8. The order of math expression evaluation will always be from left to right. The following will not work the same from machine code as BASIC:

```
10 LET Z=2+7*3
```

For clarity and compatability the above line should be rewritten:

```
10 LET Z=7*3+2
```

It is a good idea to do multiplication and division first since BASIC does them first anyway. In some cases, single expressions will have to be broken up into more than one LET statement.

9. Numbers are always in the range of 0 to 65535. No negative numbers are allowed.

Illegal statement:

```
10 LET X=-10
```

There is an exception. As long as you use only addition and

subtraction., then you can create negative numbers by subtracting them from zero. Consider the following example:

```
10 LET N=0-1
20 LET C=10
30 PRINT STR$ C
40 LET C=C+N
50 IF C=N THEN STOP
60 GOTO 30
```

The value in N will not be minus 1 but 65535 instead. Since the range of numbers is from 0 to 65535, when a variable goes below 0, it starts at the top again at 65535. It so happens that minus 1 and 65535 are the same number when added or subtracted. If you add 65535 to 10, this becomes 65545 which is beyond the range of the variable. What actually ends up in the variable is 65545-65536=9. This is the same result that we would have gotten if added minus 1 and 10.

The above exception does not work if you try to use multiplication or division.

10. There are only 26 variables A to Z. Do not use variables longer than one letter.
11. The above rule in #10 applies to strings as well.
12. Strings should usually be dimensioned at the beginning of your program. Use only numbers in a DIM () statement and do not try to redimension. String dimensioning is always optional for the highest lettered string. If you use only one string, then it is completely optional. It is up to the programmer to stay within the limits of the string.

Strings may only be one dimensional.

Illegal statement:

```
10 DIM A$ (20,20)
```

13. String statements are limited to 3 forms:

EXAMPLE I. 10 DIM A\$ (10)
 20 LET A\$="1234567"

Line 20 transfers seven characters into A\$. The last 3 characters of A\$ may be random or whatever was in the last three characters before the transfer was made.

Example II. 20 LET A\$ (10)=B\$ (20)
 30 LET C\$ (X)=D\$ (Y)

or

```
20 LET A$ (10)=INKEY$
30 LET C$ (X)=CHR$ 80
```

Only one character at a time can be transferred between strings. The brackets used above are not optional. The following are all illegal statements:

```
10 LET A$ (5to7)=B$ (9 to 11)
10 LET A$=B$
10 LET A$="JRC" (2 to 3)
10 LET A$="JRC" (3)
```

Example III.

Single letters in strings can be passed to variables one letter at a time. They can also be printed one letter at a time:

```
10 LET A= CODE A$ (N)
      or
10 PRINT A$(N)
      or
10 PRINT A$(N);
```

If you need to print out a long string, then you will have to create a loop to do so. It actually is more memory efficient to print the messages this way than to print messages ordinarily. Future versions of the compiler will probably have this problem corrected.

14. To use random numbers, you must use the expression.....

```
RND*65536
```

This isn't actually multiplication, but just a way of getting random numbers between 0 and 65535.

If you then wanted to get a random number from 1 through 10, then you would have to write it something like this.....

```
10 LET A= RND *65536/6553
20 IF 9<A THEN GOTO 10
30 LET A=A+1
```

This can be rewritten into one line.....

```
10 LET A= RND * 65536/6553+1: IF 10<A THEN GOTO 10
```

15. Inside things like A\$() or POINT (,), use only numbers or variables. Do not use expressions.
16. Beep does not work exactly the same when compiled. First of all, the numbers 65535, 65534, 65533 etc. act as negative numbers for low frequency beeps. These are the same as -1, -2, -3 etc.

In addition, the duration of the beep is 60 times quicker than what it would be in Basic. So the following would have to be rewritten.....

```
10 BEEP 1, -50
----- as -----
10 BEEP 60, 65486
```

17. There is no INPUT command handled by the compiler. However, an input subroutine is shown in the sample programs.
18. AND and OR are binary and are used solely to compare numbers on a binary basis.

ILLEGAL STATEMENT:

```
10 IF A<3 AND B<4 THEN.
```

LEGAL STATEMENT:

```
10 LET P= PEEK P AND 15.
```


The result obtained here is binary and will be completely different than what you would get if you run this in ordinary BASIC.

19. Do not put more than one comparison inside an IF, THEN statement.
20. "<" and "=" are the only comparison statements allowed. But ">" has the same effect as "<>".
WRONG: IF A>10 THEN
Right: IF 10<A THEN
21. OVER and INVERSE have no temporary values. If you use these inside a PRINT statement, they become permanent anyway.
22. INVERSE cannot be used after PLOT.
ILLEGAL STATEMENT:
10 PLOT INVERSE 1;X,Y
23. DRAW is not usable at this time. Future versions of the compiler will probably have it.
24. The BREAK key does not work when machine code is executing.
25. Do not try to compare strings directly.
WRONG: IF INKEY\$= "A" THEN
Right: IF CODE INKEY\$ = CODE "A" THEN
26. Neither FOR or NEXT are allowed.
27. Temporary colors remain in effect if the PRINT statement ends in a semicolon. Once a regular PRINT statement is used, then the permanent colors will be restored.
The same thing will happen if the PRINT statement ends in a comma or apostrophe.
28. No DATA statements are provided but it is possible to get DATA off strings.
29. No numerical arrays are provided, but it is possible to store numerical values inside strings.
30. Scientific notation is not allowed.
31. The user defined characters A to G are used to hold compiler variables and should not be used.
32. Anything not listed in the SYNTAX table is not handled by the compiler.
33. For information on how to receive updated versions of the compiler, please send a SASE to:

John Coffey
P.O. Box 448
Scottsburg, In. 47170

Please allow 3 months for a reply.

34. The compiled code is pretty much independent of the compiler itself. The compiler can be deleted and the code used only.

But if your compiled code contains the "USR" function, then it will be necessary to make one POKE before it can be used.

POKE USR "G"+4,195

35. There is an odd bug in the COPY command. After using it, you may want to use

PRINT AT 0,0; or you will get an error 5.

Section V.

THE COMPILER BASIC

The compiler is a BASIC program that is highly structured. Most of the work is done by subroutines that are in the last 2/3 of the program. Almost all of the decision making is done in the first 1/3 of the program.

Here are a few of the subroutines and what they do. Most of the routines below are variables with line numbers stored in them.

STATEMENT - This isnt actually a subroutine, but just the part that decides what to do with the next BASIC command.

LINE - Also not a subroutine. Handles the next BASIC line.

SOUND _ Converts the SOUND command to OUT.

STRING - Compiles everything that follows PRINT.

TRUE STRING - This is actually 3 subroutines to compile string handling.

POKE - Reads numbers from DATA.

CHOP - Removes first character of A\$.

ADVANCE - Advances pointer LOC for location.

FETCH - Compiles numbers.

ITEM 1 - Compiles numbers or variables.

ITEM 2 - Compiles second number or variable.

E1 - Compiles first math expression.

E2 - Compiles second math expression.

Section VI

MINIMAL BASIC AND EFFICIENT CODING

The first version of the compiler only handled these commands:

LET	GOSUB	OUT
POKE	STOP	IF
GOTO	RETURN	THEN

.....and these functions:

+	—	IN
PEEK	AND	OR
<	=	USR

I refer to the above as "minimal BASIC".

This is significant because these routines are not dependent upon the 2068 operating system. Compiled code from the above commands will run on any z80 based system assuming the memory addresses are the same.

In addition, the above commands are faster and require much less memory to be compiled in than other commands. For instance, a "GOSUB" only takes three bytes and 3 microseconds.

One part of minimal BASIC not shown above is the string handling functions. These will also run on any z80 based system and are also very memory efficient.

If you use the same commands over and over again, then you could save some memory by putting those commands inside a subroutine and using several GOSUB'S. Here is a simple example:

```
10 LET Z=10: GOSUB 1000
20 LET Z=100: GOSUB 1000
30 LET Z=1000: GOSUB 1000
40 STOP
1000 PRINT STR$ Z: RETURN
```

It is also efficient to store values in a string and have those values read by a subroutine. Here is an example using messages:

```
10 LET A$="MESSAGE 1.": GOSUB 100
20 LET A$="MESSAGE 2.": GOSUB 100
30 STOP
100 LET N=1
110 PRINT A$ (N);: IF CODE A$(N)= CODE "." THEN RETURN
120 LET N=N+1: GOTO 110
```

Here is another example with PLOT:

```
10 LET X=128: LET Y=88
20 LET A$="JKLKKLKJ."
30 GOSUB 2000
40 STOP
2000 LET P=1
2010 LET Q=P+1
2020 IF CODE A$(P)= CODE "." THEN RETURN
2030 PLOT X + CODE A$ (P)- CODE "K",Y+ CODE A$ (Q)- CODE "K"
2040 LET P=P+2: GOTO 2010
```

The letters in A\$ in the above example are used to represent X and Y coordinates. The letter K is the zero value and letters above or below it represent positive or negative coordinates. More complex figures can be drawn with the same subroutine at 2000.

Since there are only 26 values for variables, you may have to store numerical values inside strings. This is fairly easy to do if you realize that variables hold numbers from 0 to 65535 and that strings hold "numbers" from 0 to 255.

In the following program, a number is transferred from A to A\$ and B\$. It is then transferred from A\$ and B\$ to Z.

```
10 DIM A$(L): DIM B$(1)
20 LET A=2001
30 LET T=A/256: LET B$(1)= CHR$T
40 LET T=T*256: LET T=A-T
50 LET A$(1)= CHR$T
60 LET Z=256* CODE B$(1)+ CODE A$(1)
70 PRINT STR$ Z
```

Note the usage of a temporary variable called T. I used this to break up expressions too complex for the compiler to handle at one time.

SECTION VII
*****COMPILER TEST PROGRAMS*****

```
5 LET A$="ABC"
10 LET A$(2)=A$(3)
20 PRINT A$(1);A$(2);A$(3);
8999 STOP
9000: DIM S(25); LET M=0: LET MO
```

```
100 DIM Z$(100)
200 LET Z$="THIS WILL BE PRINTED
BACKWARDS!"
300 LET P=1
400 IF CODE Z$(P)-CODE "I" THEN
LET P=P+1: GO TO 400: REM NOTE
THAT "-" IS THE SAME AS "<"
500 PRINT Z$(P); LET P=P-1: IF
P THEN GO TO 500: REM GOTO 500
UNLESS ZERO
```

10 REM THIS PROGRAM WILL CONVE
RT EVERY CHARACTER ON SCREEN TO
ITS INVERSE

```
20 LET A=16384: LET B=1024*8
30 POKE A,255-PEEK A: LET A=A+
1: LET B=B-1: IF B THEN GO TO 30
8999 STOP
```

10 REM THIS PROGRAM USES THE
ARROW KEYS TO DRAW LINES UNTIL
YOU GO OFF THE SCREEN.

```
20 LET X=128: LET Y=86
30 LET K=CODE INKEY$
40 IF K=CODE "S" THEN LET X=X-
1
50 IF K=CODE "S" THEN LET Y=Y-
1
60 IF K=CODE "7" THEN LET Y=Y+
1
70 IF K=CODE "8" THEN LET X=X+
1
80 PAUSE 1: PLOT X,Y: GO TO 30
8999 STOP
```

```
9000: DIM S(25); LET M=0: LET MO
```

10 REM THIS PROGRAM INPUTS 2
NUMBERS AND PRINTS THE
DIVIDED RESULT

```
20 PRINT AT 8,10;
30 GO SUB 1000: LET A=Z
40 PRINT AT 8,10;" ";AT 9,9;" /
";
50 GO SUB 1000: LET B=Z
60 PRINT AT 9,10;" ";AT 10,10;
"-----"
70 LET C=A/B: PRINT AT 11,11;S
TR$ C
80 REM TIME TO QUIT
90 STOP
1000 LET Z=0
1010 PRINT FLASH 1;" "; FLASH 0;
1020 PAUSE 0
1030 BEEP 1,0
1040 LET K=CODE INKEY$: IF K=13
THEN RETURN
1045 IF K<CODE "0" THEN GO TO 10
20
1050 IF CODE "9"<K THEN GO TO 10
20
1055 PRINT CHR$ K;
1060 LET Z=Z*10+K-CODE "0"
1070 GO TO 1020
8999 STOP
```

THE ABOVE SUBROUTINE AT LINE
1000 WILL INPUT A NUMBER INTO
THE VARIABLE Z

10 REM THIS PROGRAM SHOWS HOW
TO USE THE JOYSTICKS BY
USING IN AND OUT.

```
20 OUT 245,14: LET J=IN 4598:
PRINT "STICK 1": GO SUB 100
30 REM
40 OUT 245,14: LET J=IN 4854:
PRINT "STICK 2": GO SUB 100
50 REM
60 OUT 245,14: LET J=IN 5110:
PRINT "EITHER STICK": GO SUB 100
70 REM
80 IF CODE INKEY$=CODE " " THE
N STOP
83 REM
85 GO TO 10
90 REM
100 IF J<128 THEN PRINT "THE BU
TTON IS PRESSED": LET J=J+128
110 IF J<255 THEN PRINT "THE VA
LUE IS "; LET J=255-J: PRINT ST
R$ J
120 RETURN
8999 STOP
```

10 REM THIS PROGRAM SHOWS A
USE OF "OR" TO CREATE
A DARKER SCREEN. NOTE
THAT "OR" BEHAVES DIF-
FERENTLY IN MACHINE
CODE. IT IS A BINARY
OR INSTEAD OF BOOLEAN.
THIS PROGRAM WILL NOT
WORK IN ORDINARY BASIC.

```
20 LET S=16384: LET C=6144
30 POKE S,PEEK S+PEEK S OR PEE
K S
40 LET S=S+1: LET C=C-1
50 IF C THEN GO TO 30
8999 STOP
```

```
10 REM CLOCK
20 RANDOMIZE 0
30 LET R=AND*65536: PRINT STR$
R
35 POKE 23692,99
40 IF CODE INKEY$=CODE " " THE
N GO TO 20
50 REM "-" IS SAME AS "<"
8999 STOP
```

10 REM THIS PROGRAM SHOWS HOW
TO GET A RANDOM NUMBER
FROM 0 TO 9
15 REM THIS WILL WORK A LITTLE
DIFFERENT IN BASIC THAN
MACHINE CODE

```
20 LET R=AND*65536/6553: IF 9<
R THEN GO TO 20
30 PRINT STR$ R
40 GO TO 20
8999 STOP
```

5 REM NEXT LINE WILL LOOK ODD
10 SOUND 7,62;8,16;1,1;12,20;1
3,12
20 PAUSE 200
30 REM TO TURN OFF RAPPING
SOUND USE: SOUND 8,0
8999 STOP

```
10 REM ***RANDOM NUMBERS***
25 LET B=20
50 LET A=AND*65536
100 LPRINT TAB 10;"VARIABLE A =
";STR$ A
200 LET B=B-1: IF B THEN GO TO
50
8999 STOP
```